Comparison of Serial and OpenMP Parallel Performance in Vector Matrix Multiplication

Ben Klein
Computer Science Department
San Francisco State University
San Francisco, CA, USA
Email: bklein@sfsu.edu

***Abstract*—This study looks at performance of four implementations of vector matrix multiplication code. The versions are: a basic serial version, a vectorized version utilizing compiler flags, an OpenMP parallel version with multiple thread size tests, and a reference CBLAS version. Every code was ran on a Perlmutter CPU node on problem sizes ranging from 1024 to 16484. Results show that compiler vectorization has massive gains over basic code, OpenMP parallel code suffers a performance hit at small problem sizes with large thread counts, and the ideal thread count for most problem sizes tested is four.**

I. Introduction

Vector matrix multiplication (VMM) is the foundation for many modern domains of computing and engineering. The runtime of the algorithm is highly dependent on implementation, which means that building a solid implementation will result in easy performance gains for many applications. One method of improving matrix multiplication performance is parallelization, which is the focus of this study.

This study compares the performance of four different algorithms: a straight-forward VMM algorithm, a vectorized version of that algorithm, an OpenMP parallelized version, and the C Basic Linear Algebra Subprograms (CBLAS) version used as a reference. This study tests the codes on five different problem sizes and four different numbers of threads on the Perlmutter CPU nodes (More information on the computational environment in Section III). In Section II the details of the code implementations are listed.

The results show that the parallelized OpenMP version performs best as problem sizes increase, up to a limit. At smaller problem sizes, serial code, like the vectorized VMM and CBLAS perform better, but fall off as problem sizes increase. This hints at an overhead cost to OpenMP that causes performance hits in small problems but is more than made up for as the problem size scales up.

II. Implementation

This section describes the four implementations of vector matrix multiplication (VMM) compared in this study. Each version performs the same operation: $y := A * x + y$ however, the performance differs. In the first subsection, the basic serial implementation is detailed. The next subsection shows a vectorized version that uses compiler flags to make performance gains on the same basic code. Then, the CBLAS implementation is briefly introduced, which will serve as the baseline for comparison. Finally, the OpenMP implementation is shown, which operates over rows in parallel threads.

A. Basic Vector Matrix Multiplication

The basic implementation follows the form: $y := A * x + y$ and computes the dot product between a row of A and a vector x. The code performs serially on one thread and does not have flags for vectorization during compile time. The code is shown in Listing 1.

```
1 // Vector Matrix Multiplication: Y :=  A * X +
    Y
2 // where A is n-by-n matrix stored in row-
    major format, and X and Y are n by 1
    vectors.
3 // On exit, A and X maintain their input
    values.
4 void VMM_basic(int n, double *A, double *x,
    double *y) {
5     for (int i = 0; i < n; i++) {
6         sum = 0;
7         for (int j = 0; j < n; j++) {
8             sum += A[i * n + j] * x[j];
9         }
10        y[i] = sum;
11    }
12 }
```

Listing 1. Vector Matrix Multiplication: Y := A * X + Y.

B. Vectorized Vector Matrix Multiplication

The vectorized version is the exact same code as the previous basic implementation shown in Listing 1. The difference is that this code is compiled with the following compile flags:

```
-O3 -DNDEBUG -Wall
-fomit-frame-pointer -ftree-vectorize
-funroll-loops -ffast-math
-fopt-info-vec-all=report.txt
```

C. C Basic Linear Algebra Subprograms (CBLAS)

The CBLAS implementation of VMM performs the same operation as in the previous subsections, and is used as the serial baseline for comparison to the other codes in this study. The implementation will be treated as a black box for the purposes of this study but the documentation can be found at https://www.netlib.org/blas/ [1].

D. OpenMP

This version parallelizes the previous code by using OpenMP to create threads that operate on each row. The `#pragma omp simd reduction(+:sum)` line helps the compiler vectorize the inner loop.

in Listing 2.

```
void OpenMP_VMM(int n, double *A, double *x,
    double *y)
{
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for (int j = 0; j < n; j++)
        {
            sum += A[i * n + j] * x[j];
        }
        y[i] += sum;
    }
}
```

Listing 2. OpenMP VMM parallelized code, with each thread operating on different rows.

III. Results

The following section details the evaluation results for performance for the different implementations of VMM. First the computational environment and compiler flags are described, followed by the test methodology and metrics used for evaluation. Then the basic, vectorized and CBLAS versions are compared in terms of MFLOP/s. Next, the speedup of the parallel OpenMP code is analyzed. Finally, there is a comparison of OpenMP's best performing thread count code with CBLAS, and a discussion around bandwidth utilization as a percentage of Perlmutter's peak.

A. Computational Platform and Software Environment

The performance studies were conducted on a CPU node of the Perlmutter system at NERSC. [2] Each node has two AMD EPYC 7763 processors running at 2.45 GHz. Each core has a 32 KB L1 cache (with a 64-byte cache line size) and a 512 KB L2 cache, while all eight cores in a CCD share a 32 MB L3 cache. The node has 512 GB of memory, with a measured peak memory bandwidth of 204.8 GB/s.

The software environment consisted of the GNU C++14 compiler (SUSE Linux) version 7.5.0 on SUSE Linux Enterprise Server 15 SP5. The build system used `cmake` version 3.20.4 and `make` version 4.2.1. The vectorized code used `-O3 -DNDEBUG -Wall -fomit-frame-pointer -ftree-vectorize -funroll-loops -ffast-math -fopt-info-vec-all=report.txt` and OpenMP code used static scheduling.

B. Test Methodology

The correctness of each implementation was compared to the baseline CBLAS implementation for verification. The primary performance metric used is MFLOP/s. In the vector matrix multiplication operation: $y := A*x+y$, each of the n rows performs n multiplications and n additions, resulting in a total FLOP count of $2n^2$. From there we can derive the MFLOP/s rate as:

$$\text{MFLOP/s} = \frac{2n^2}{t \times 10^6}$$

where t is runtime in seconds.

The other metric used is speedup, defined as:

$$S(n,p) = \frac{T^*(n)}{T(n,p)},$$

where $T^*(n)$ is the baseline serial runtime, $T(n,p)$ is the runtime using p threads. This metric is used to show the gains that parallelization yields in practice, as actual parallel performance gains are almost never at the ideal limit that perfect parallelization would allow.

The problem sizes tested are

$$\{1024, 2048, 4096, 8192, 16384\}$$

and the thread counts tested in the OpenMP code are $\{1, 4, 16, 64\}$.

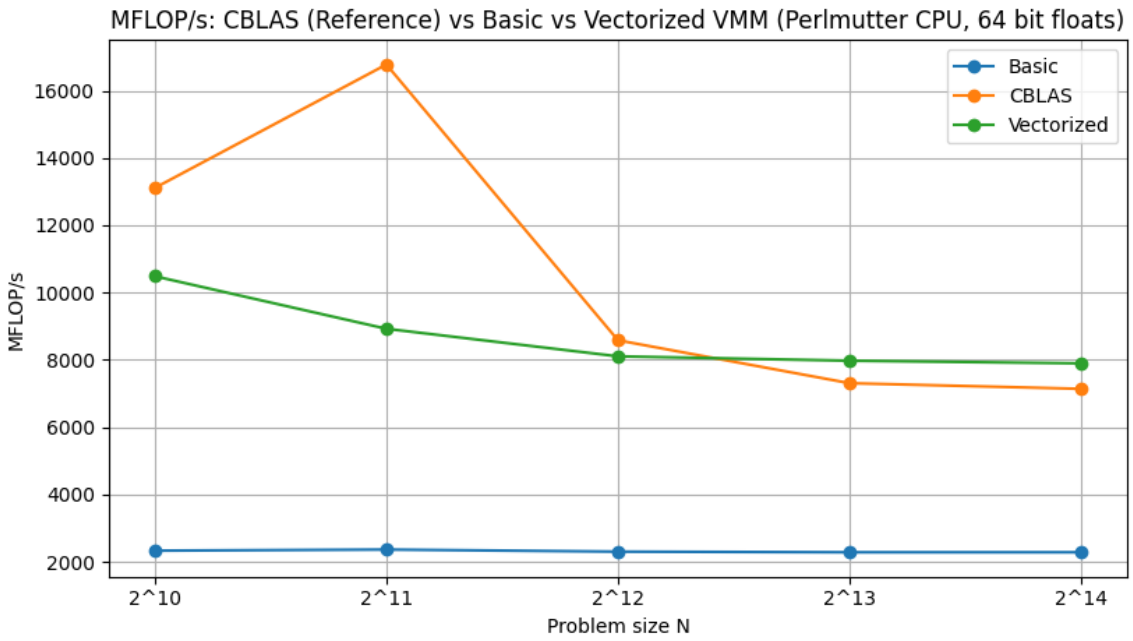

Fig. 1. MFLOP/s comparison of Basic, Vectorized, and CBLAS implementations.

```
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:11:22: missed: couldn't vectorize loop
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:16:28: missed: not vectorized: complicated access pattern.
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:14:25: optimized: loop vectorized using 32 byte vectors
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:14:25: optimized: loop vectorized using 16 byte vectors
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:9:6: note: vectorized 1 loops in function.
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Analysis failed with vector mode V4DF
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** The result for vector mode V32QI would be the same
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Re-trying analysis with vector mode V16QI
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Analysis failed with vector mode V16QI
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Re-trying analysis with vector mode V8QI
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Analysis failed with vector mode V8QI
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Re-trying analysis with vector mode V4QI
/global/homes/b/bklein/cp_3/dgemv-vectorized.cpp:18:12: note: ***** Analysis failed with vector mode V4QI
report.txt lines 1-13/13 (END)
```

Fig. 2. Vectorization report on Perlmutter. 32 byte vectors imply utilization of the AVX2 instruction set.

C. CBLAS vs. Basic VMM vs. Vectorized VMM

Figure 1 shows the MFLOP/s of the basic, vectorized and CBLAS implementations on all problem sizes.

The basic implementation shows a very consistent, poor performance at a little over 2000 MFLOP/s on average. In comparison, the vectorized VMM uses the same code and performs 4-5 times faster with compiler flags that allow for vectorization. Figure 2 shows the vectorization report from Perlmutter. Vectorization reports can be terse and include a lot of information, but the loop vectorization in 32 bytes implies that the vectorized code is making usage of the AVX2 instruction set. This means the CPU is doing 256-bit wide SIMD operations. As far as scaling over problem sizes, the vectorized code, works best on small sizes at a little over 10,000 MFLOP/s and falls off a little as problem sizes increase ending around 8000 MFLOP/s at a problem size of 16384.

The CBLAS version is more dynamic in it's performance across problem sizes. It peaks at the 2048 problem size at roughly 17,000 MFLOP/s but ends even lower than the vectorized code, a little over 7000 MFLOP/s at problem size 16384. This likely points at memory optimizations for a particular cache and problem size that don't scale well as the memory overflows the cache lines.

D. Evaluation of OpenMP Parallel VMM

This subsection evaluates the way that performance scales across both problem sizes and numbers of parallel ranks in OpenMP VMM code. Using the speedup metric: $S(n,p) = T^*(n)/T(n,p)$, the baseline serial code: $T^*(n)$ is the OpenMP 1 thread runtime. This baseline is chosen to be as similar as possible to the multithreaded

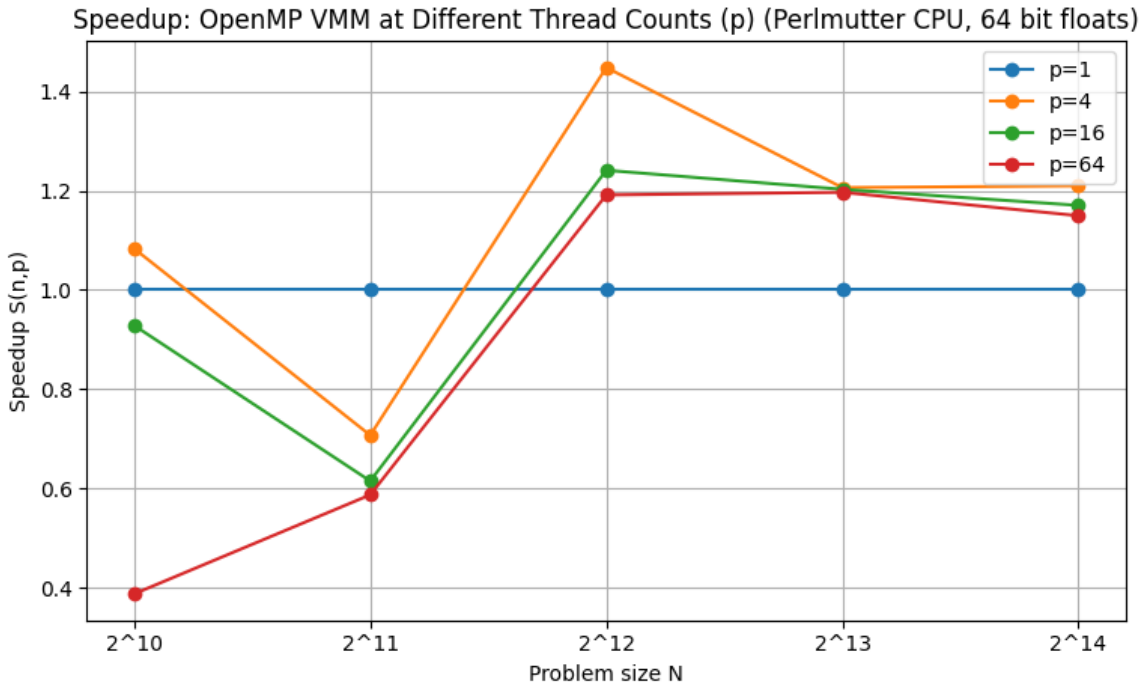

Fig. 3. Speedup of OpenMP VMM with static thread scheduling. The serial baseline for the speedup calculation is the OpenMP 1 thread runtime.

code as possible. All runs use a static thread schedule. Figure 3 shows the speedup of all thread and problem sizes.

As the baseline is defined as OpenMP 1 thread runtime, the 1 thread speedup is a flat line. The rest of the thread sizes at 4, 16, and 64 generally follow a similar trend, dipping to their lowest points in small problem sizes at either 1024 or 2048, and reaching a peak speedup of 1.2 to 1.4 at problem size 4096. As the problem sizes scale further, the speedup seems to flatten out around a speedup value of 1.2. The best performing thread count is consistently 4 threads, which implies an overhead cost to creating too many threads. This is reinforced by the fact that at small problem sizes, the 64 thread code performs far worse, and is likely slowed down by thread management for a small task.

E. Comparison of OpenMP VMM with CBLAS

Figure 4 compares MFLOP/s of the best performing OpenMP thread size (p=4) to the serial CBLAS code. The performance trends are nearly opposite: as CBLAS hits it's peak 17,000 MFLOP/s at 2048 problem size, OpenMP at four threads is just under 6000 MFLOP/s. Then at problem size 4096, OpenMP starts outperforming as the serial CBLAS code performance drops sharply down to under 9000 MFLOP/s.

CBLAS performs well on small problem sizes likely because of optimizations for small cache sizes, but it falls off as the data overflows the cache. The OpenMP implementation in contrast, seems to struggle at smaller problem sizes as the overhead of thread management overshadows the tiny amount of work being done. But as the problem size grows, more work can be shared across threads, and performance climbs again.

F. Overall Findings and Discussion

In Table I bandwidth utilization across multiple problem sizes is shown.

To determine bandwidth utilization, we need to look at bandwidth first. Each matrix is $n \times n$, and we are multiplying by an n sized array into an n sized array. Multiplying each term by 8 bytes for the size of a double results in a total bandwidth usage of $8n^2 + 16n$. Perlmutter CPU nodes have a measured peak memory bandwidth of 204.8 GB/s, so the peak bandwidth is calculated as a percentage.

CBLAS and the vectorized code use the memory bandwidth effectively at small problem sizes, which reflects a trend for optimized serial code. For the OpenMP code, the memory bandwidth tends to be optimal around 4096 problem size. At 64 threads, at small problem sizes

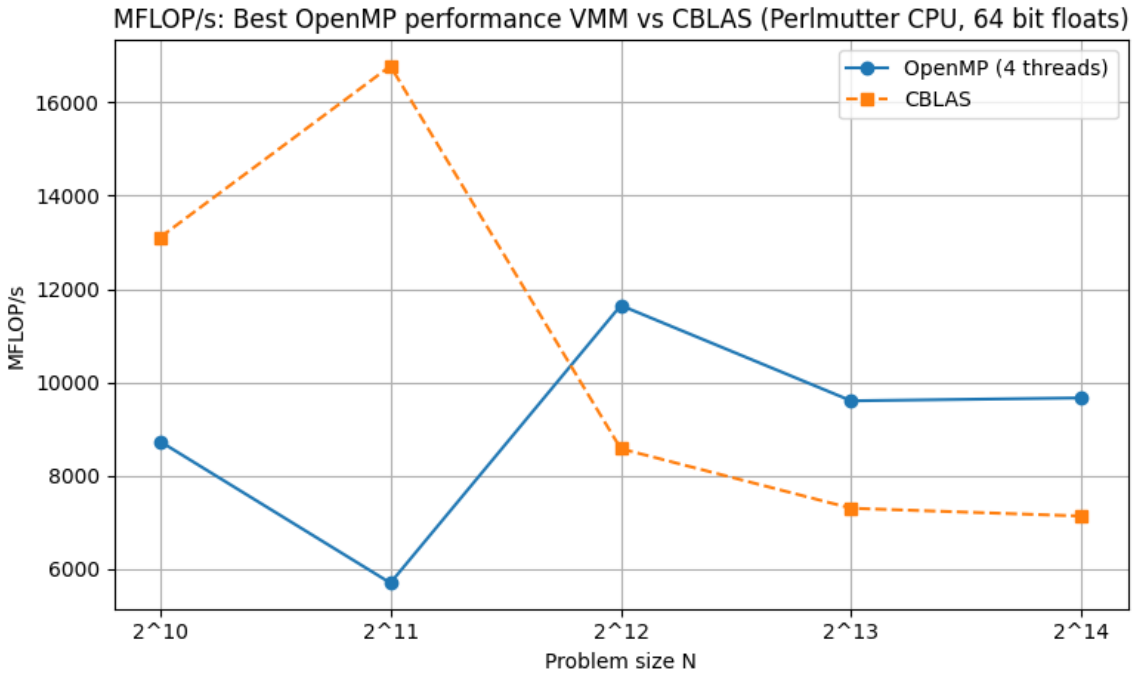

Fig. 4. MFLOP/s: best OpenMP-parallel VMM (p=4) vs. serial CBLAS.

TABLE I
ESTIMATED % OF PEAK MEMORY BANDWIDTH UTILIZATION ON PERLMUTTER (PEAK = 204.8 GB/S).

Size (n)	Basic	CBLAS	OMP-1	OMP-4	OMP-16	OMP-64	Vectorized
1024	4.6%	25.6%	15.8%	17.1%	14.7%	6.1%	20.5%
2048	4.6%	32.8%	15.8%	11.2%	9.7%	9.3%	17.4%
4096	4.5%	16.8%	15.7%	22.8%	19.5%	18.7%	15.8%
8192	4.5%	14.3%	15.6%	18.8%	18.7%	18.6%	15.6%
16384	4.5%	13.9%	15.6%	18.9%	18.3%	18.0%	15.4%

the bandwidth usage becomes very poor as adding extra threads doesn't seem to speedup the performance.

Four threads seems to be the sweet spot for OpenMP performance, which is the right balance between overhead with thread management and parallel speedrun gains. The worst is 64 threads for small problems, because of the thread overhead, and the worst for big problems is the 1 thread OpenMP code, simply because there is no parallel performance gain.

ACKNOWLEDGEMENT

The author acknowledges the use of OpenAI's ChatGPT 5 for assistance in graphing, formatting, and proofreading.

REFERENCES

[1] "BLAS (Basic Linear Algebra Subprograms)." [Online]. Available: https://www.netlib.org/blas/

[2] "Architecture - NERSC Documentation." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/